ICAM Development Standards

- All web requests shall follow the MVC pattern of Controller -> Model -> View. All actions need to use a Controller (including Ajax/JSON requests).
- Twitter bootstrap shall be used for the UI and UX components of the web app.
    - http://twitter.github.io/bootstrap/
    - Tables should not be used for layout, Grid system of twtter bootstrap should be utilized.
- Jquery shall be used for the javascript framework of the web app.
    - http://jquery.com
- Models shall be used to represent data objects that reside in LDAP and the Database. Hashmaps should not be used, however, exceptions to this would be expected.
- Any Java file checked in to trunk shall be free of error and ***warning*** messages.
- Standard Java conventions shall be used for naming files and methods.
    - Classes start with a capital letter (camel case).
    - Class name should be a noun.
    - Methods start with lower case.
    - Name of the method should imply an action.
    - Variables should start with lower case.
- Java applets should not be used, whenever possible, due to security concerns. If a developer needs to create a java applet on the client side, this should be vetted through the ICAM development team.
- Log4J shall be used for all logging actions.
- Exceptions shall be written to log4j and the exception table in the icam database.
- The java “new” operator shall not be used to create an instance of a class, unless that class is in the current project or the Common project.
- Only Interfaces shall be used to interact between separate projects.
- Interfaces shall only throw IdmaxExceptions.
- A width of 1280px shall be used for UI layout.
- See Asset file structure.
- See View file structure.
- See Java Package Conventions.
- For SVN project layout see “ICAM Dev SVN Project Layouts.docx”
- For High Level code release strategy see “ICAM Dev SVN Workflows.docx”
- For Details on how to create release branches in SVN and perform the builds in bamboo for the release see “SVN-Bamboo-branching.docx”

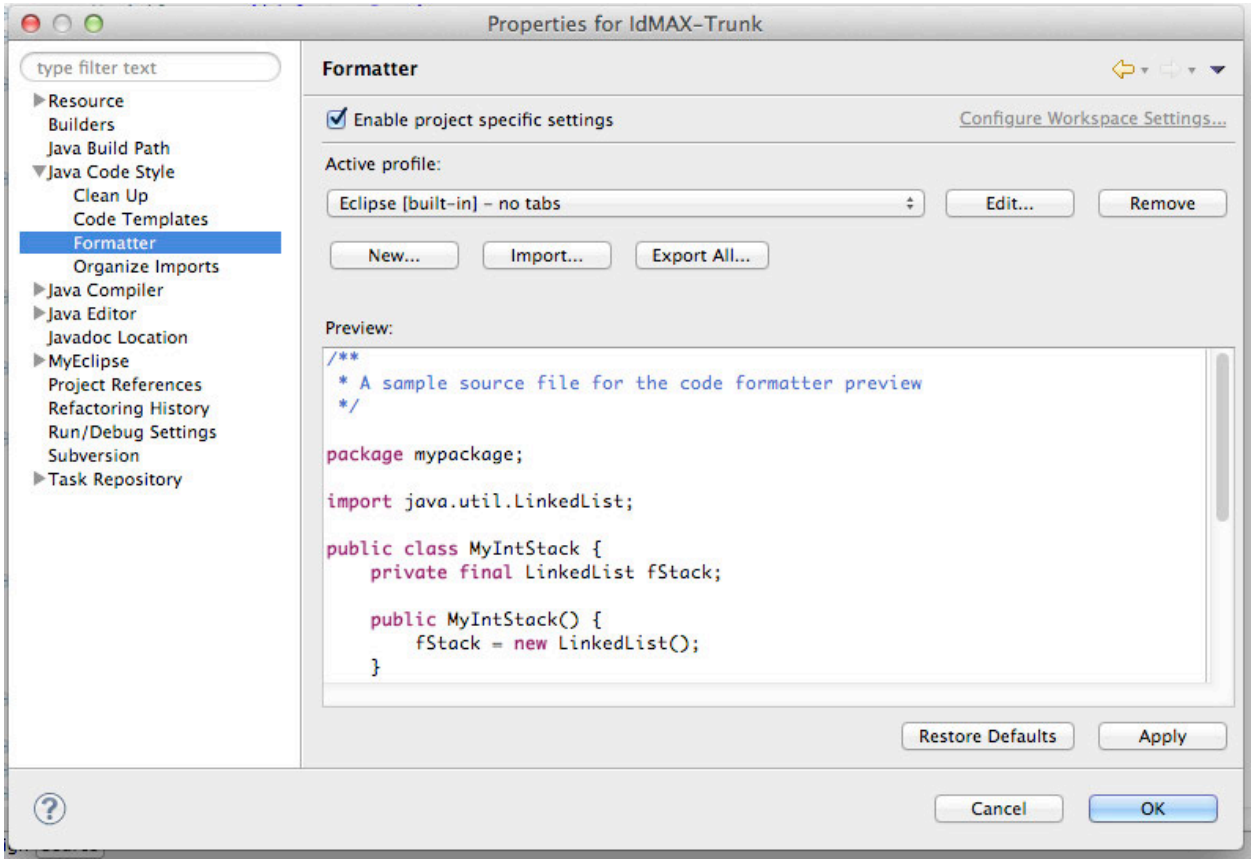
BEST PRACTICES

1. Methods should be 20 lines or less – less is better. “The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than

that. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long."

2. Use descriptive ("intention revealing") names for methods – "You know you are working on clean code when each routine turns out to be pretty much what you expected.

3. Classes: smaller is better and a class should try to address one issue and one issue only. When you get over 500/1000 lines in a class it is getting pretty hard to understand what the purpose of the class is. It probably should be refactored.
4. Avoid inner classes.
5. Limit the accessibility of methods and fields – default to private - http://www.oracle.com/technetwork/java/seccodeguide-139067.html
6. Use 4 spaces for indentions instead of tabs. Tabs wont show up the same in all editors.



7. Package names should only be in lowercase - http://ace.apache.org/dev-doc/coding-standards.html
8. Always return empty collections and arrays instead of null - http://viralpatel.net/blogs/most-useful-java-best-practice-quotes-java-developers/

9. Use Java constants where appropriate, instead of using hard coded values.

unit testing:

10. Write integration tests for both a successful case and a failure case and a null value case (at least) for all public Controller methods.
11. Write integration tests for both a successful case and a failure case and a null value case (at least) for all Interface methods.
12. Write unit tests for both a successful case and a failure case and a null value case (at least) for all utility class (ex. DateUtilities) public methods.

Java Package Conventions:

The common denominator is that the packages start with "gov.nasa.idmax" and then add the project name.

Examples:
Idmax3Business (Project)
- gov.nasa.idmax.business.constants
- gov.nasa.idmax.business.implementations
- gov.nasa.idmax.business.utilities
- …

Idmax3Common (Project)
- gov.nasa.idmax.common.constants
- gov.nasa.idmax.common.domain (domain objects are the models)
- gov.nasa.idmax.common.interfaces
- gov.nasa.idmax.common.utilities
- …

Idmax3Web (Project)
- gov.nasa.idmax.web.controllers
- gov.nasa.idmax.web.utilities
- …

For the Public projects:

The common denominator is that the packages start with “gov.nasa.icampublic” + the project name. And for the individual web projects the web project name (“USS”) is added.

ICAMPublicCommon (Project)
  gov.nasa.icampublic.common.domain
  …

ICAMPublicDao(Project)
  gov.nasa.icampublic.dao.implentations
  …

USSPublicBusiness
  gov.nasa.icampublic.uss.business.implementations
  …

USSPublicWeb
  gov.nasa.icampublic.uss.web.controllers
  ….

Asset File Structure

What are assets?

Assets are the JavaScript, CSS, and Images needed to support your HTML views. This should all be placed under a folder called “assets”. Any asset that is applied to the web app overall should be placed at the root of that assets folder. Any asset specific to a controller and its actions should be placed within a folder named after the controller. See example below (this should mirror bootstraps layout):

```
assets/
        js/
                Jquery.js
                Bootstrap.js
                Termination/
                        Termination.js
                Identity/
                        Identity.js
        css/
                Bootstrap.css
                Idmax.css
```

```
            Termination/
                Termination.css
    img/
```

Views File Structure

All views shall be placed under a folder called views and the same conventions apply as the assets. Views that apply to the overall web application should be placed at the root of views folder. Any views related to a specific controller shall be placed under a folder with the name similar to the controller. This should also match the url that the webapp goes to, by convention, any other developer knows where to look to find the jsp. E.G. if you have a url of /nams/user there should be corresponding folders under views called /nams/user.

```
Views/
    Layout.jsp
    Termination/
        Index.jsp
        …
    Identity/
        Index.jsp
        …
```

Validations

Use parsley.js for client side validation.

http://parsleyjs.org/documentation.html

For server side use the Spring <form:errors path="*"/> at the top of the form. You can reference this in the training material from the Spring class. Or look it up online.

For hard errors use the errorMessage field at the top of the layout.jsp. This would generally be used when you encounter an error on the server side in the business logic or something unexpected that you can't necessarily tie back to the form itself.

Forms should always have server-side validation to make sure data integrity stays intact.

Development Pattern for Forms - GET to POST

This section is to provide guidance on creating a Data Transfer Object (DTO) that can represent your data in the form. This is usually comprised of 1 or more Domain Objects (DO). Another way to think of this is that a DTO represents your Form object and the DO represents the back-end data.

During the Spring MVC training we learned multiple best practices that we need to incorporate into our development efforts. One of those was to keep the web side decoupled from the back-end database tables, which this accomplishes. It is hoped that this will provide those on the team who are just starting to have a place to start from, so they don't have to reinvent the wheel each time. This will provide a level of consistency throughout the web app and will hopefully allow developers to move around to different parts of the web app without being completely lost.

Here's a general outline of what you will need to do in order to GET data, display it to the user, have the user make some changes, and then POST it back to your controller to save the changes.

1. Need to define a Controller to handle your methods for the GET and the POST.
2. In the GET, you will need to populate a model (DTO) that the Form can use to display the information you received from the service layer.
3. The Form will be submitted back to the controller using a POST method.
    a. The POST method will take a parameter that represents the Form as a DTO.
    b. This will automatically bind what was on the Form to the DTO.
    c. A validate() method will be called on the DTO to validate the form data.
    d. The DTO will have methods to convert back to the DOs. The DOs can then be passed down to the service layer to perform the update on the back-end.

The DTO will need to contain the following elements:

- A constructor that takes in the necessary DOs needed to create the Form.
- to methods to convert the DTO back to the necessary DOs.
    - These should be named to{DO name}(). E.G. If I was converting elements from a Form to a NamsApplication DO, it would be called **public NamsApplication toNamsApplication(){}**
- A **public void validate(Error errors) {}** method to perform your server-side validation on the Form.
- Getters and Setters for the form elements that are displayed on the screen.
    - This will allow both the display of those elements after the GET and also the binding back to the DTO during the POST.
- These DTO (or Form objects) should be placed in the Idmax3Web project in the package **gov.nasa.idmax.web.domain.{controller}.** Within this package you can create all the Form objects that are necessary for your controller.

See the NamsApplicationForm.java in the gov.nasa.idmax.web.domain.wct package as an example.

This is just a pattern. It is understood that there will be exceptions to this pattern and that is ok. In normal circumstances, please try and follow the pattern outlined above so we can stay consistent.